# Microcontroller ASIC Design Group
# SD0923

---

Members: Adam Hoffert

Tom Narvesson

Zachary Skalsky

Filipe Betzel

# Table of Contents

## Project Overview

The purpose of our project was to create a custom microcontroller using an open source processor core and three custom peripherals designed, tested and build by our team. Our sponsor was Packet Digital LLC, and their goal was to help our group gain an understanding of the digital design process. During the year, we gained a lot of experience in many different areas of electrical engineering.

First, we had to learn about hardware definition languages. All of our peripherals were written in Verilog HDL, and we used the Incisive Unified Simulator from Cadence and ModelSim from Mentor Graphics to do all of our simulation. Both tools are used in industry for simulating Verilog and VHDL. We also got experience in using the Linux operating system, since Incisive ran exclusively on Linux. We also learned a lot about the synthesis process, and FPGAs. We used Altera Quartus and an Altera Cyclone 2 FPGA for all of our functional testing, and learned about a lot of the quirks involved with simulation versus synthesis.

We also learned a lot about designing and building modules that have to interface with other peoples designs. There were some problems with having four different people, each with their own ideas and methods building modules, but by second semester we had learned how to best manage our time and work around our group members. Our project is rather unique in that we had to define our hardware using Verilog, write software in C that ran on the MSP430, and also write a PC application in C++ to handle the PC to microcontroller data transfers. This probably doesn't make sense right now, but hopefully reading the rest of this document will clarify it.

Finally, we had to build a demo system that tested and showcased all of our accomplishments. A PCB was fabricated, and we had to make sure that the parts and components on it worked according to our specifications, as well as the components manufacturer specs. Software was written to perform some basic tasks, and a useful product concept was built.

We think that we learned a lot by doing this project, and we hope that someone will be able to either finish what we have started and make a full chip, or somehow be able to use our microprocessor for some sort of useful task.

## Demo System

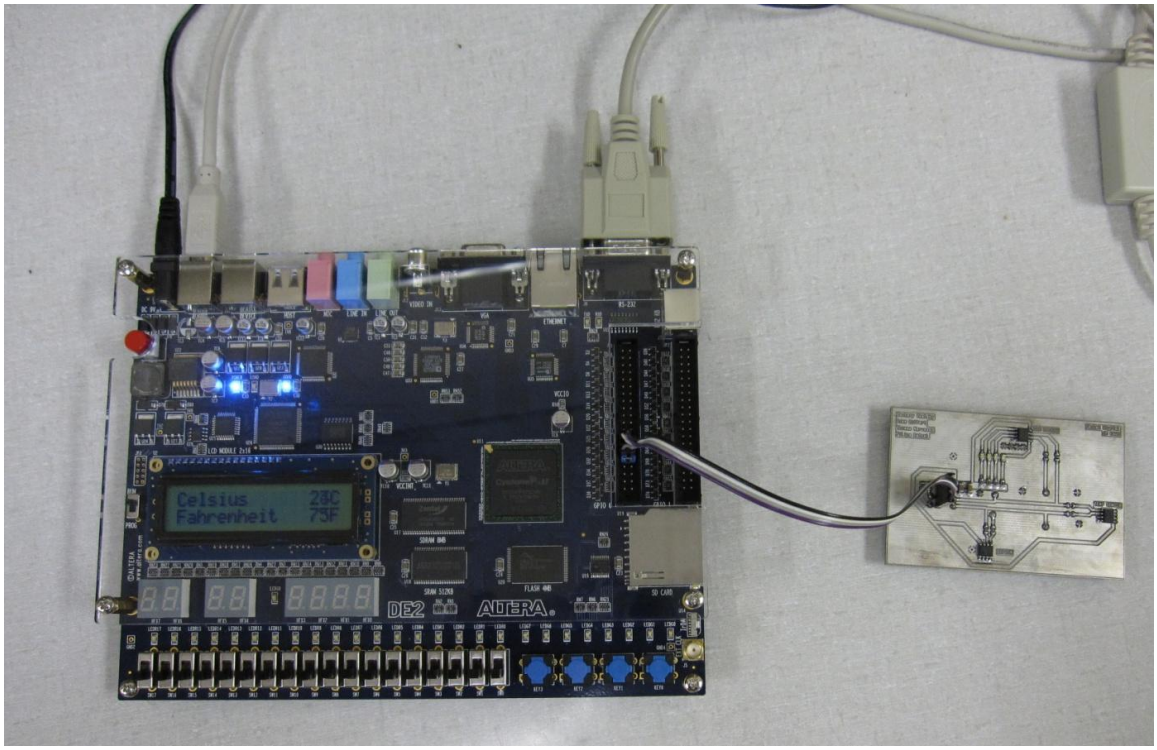



Figure 1

In Figure 1 we can see the completed demo system. Our synthesized Verilog along with the MSP430 is running on the Altera DE2 FPGA evaluation kit. Our demo PCB is connected to the microcontroller through the multicolored cable running off of the right side of the DE2. The cable carries the power, ground for the PCB as well as the sclk and sdat signals. The SMBus pullup resistors are located on the PCB. On top of the DE2, we can see the serial port that connects the PC to the DE2. There is also a black power cord and a tan USB programming cable plugged in on the left side. The last area of interest is the LCD, which is displaying the current temperature of the temp sensor.

# Schematics

## Top level block diagram

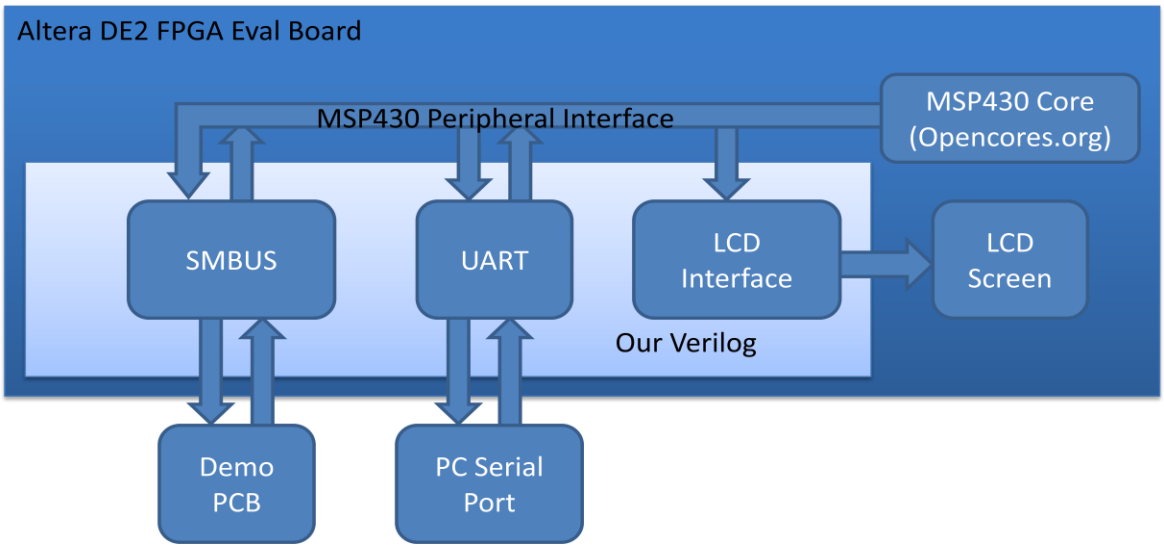



Figure 2

This is the overall block diagram of our demo system. Everything contained inside the large blue box is what is located on the DE2. As we can see from Figure 2, the LCD, as well as our microcontroller is running on the DE2. The blocks inside of the light blue box are what we designed ourselves. They are interfaced to the MSP430 through the peripheral bus.

## Peripheral Block Diagrams

### UART

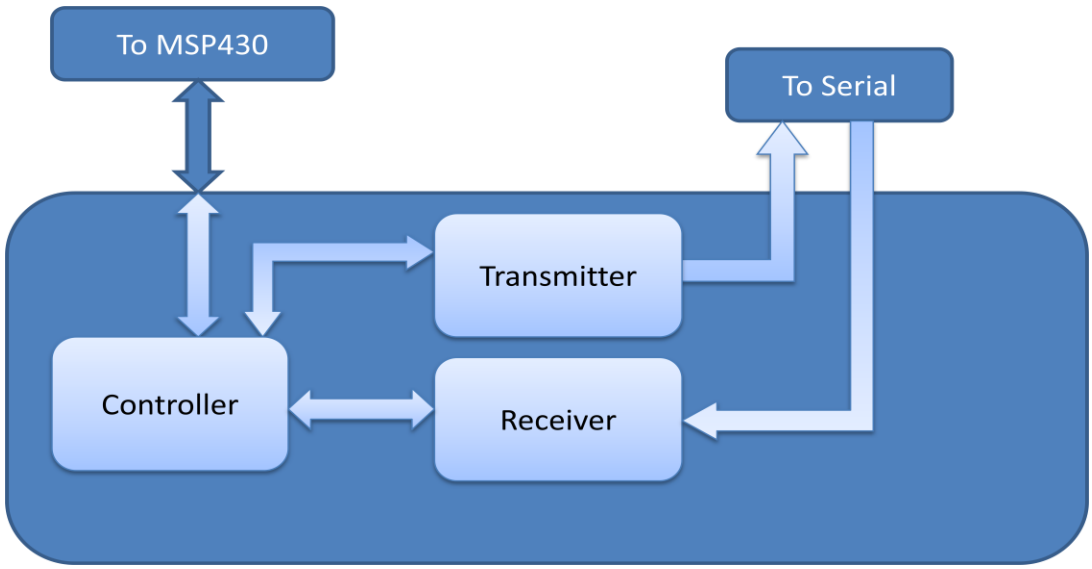



Figure 3

## LCD Interface

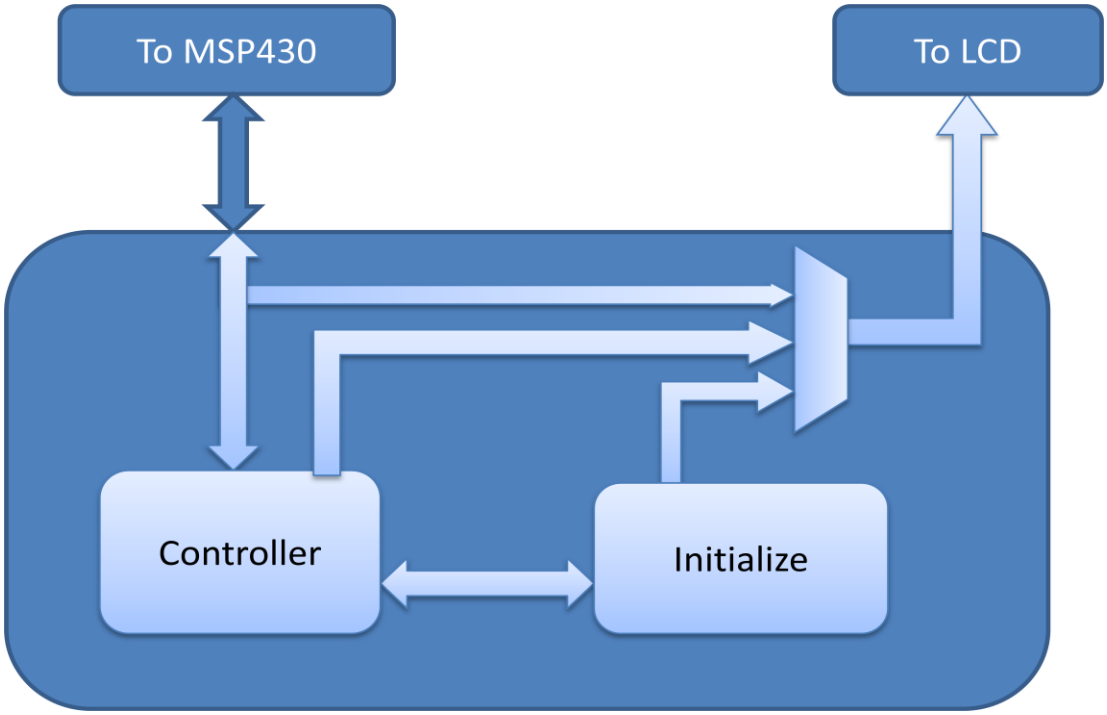



Figure 4

## SMBUS

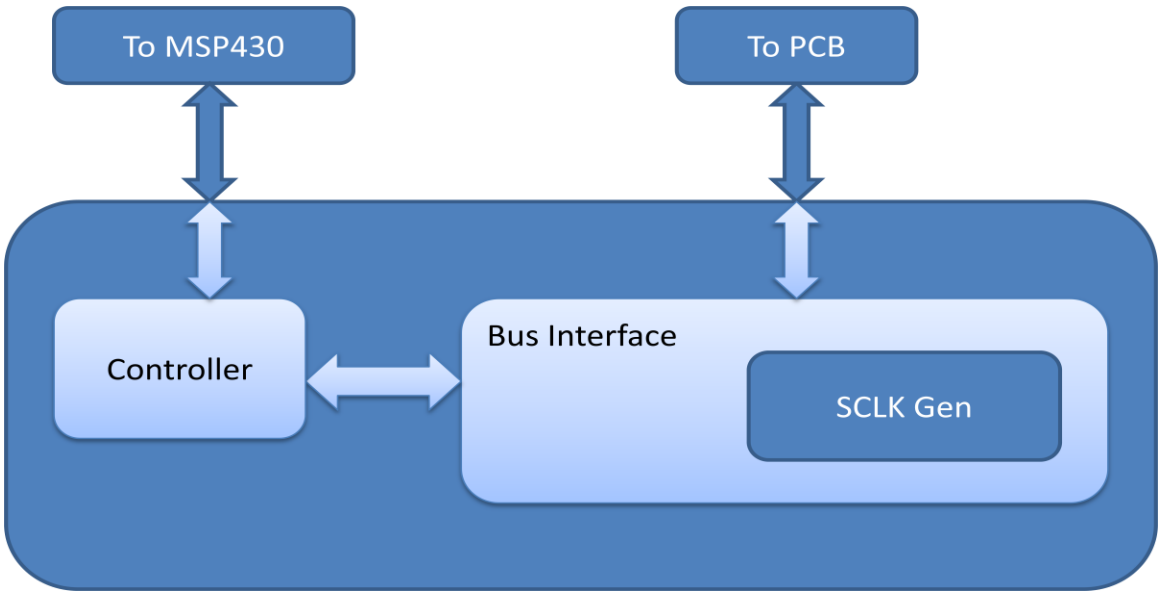



Figure 5

# RTL

RTL stands for Register Transfer Level, and is a way of describing the operation of a digital circuit. It is what is used in a hardware definition language to make circuits from code. We chose to write all of our peripherals in Verilog HDL due to the fact that the OpenMPS430 was built in Verilog. Included is a brief description of ports, modules and configurable files that may be of interest to the end user.

## UART

### Theory of Operation

UART stands for Universal Asynchronous Receiver/Transmitter. It uses a single transmit wire and a single receive wire and can communicate with one other UART. The UART works by taking a byte of data and transmitting it sequentially bit by bit. The transmission scheme that our part uses is displayed in Figure 6.

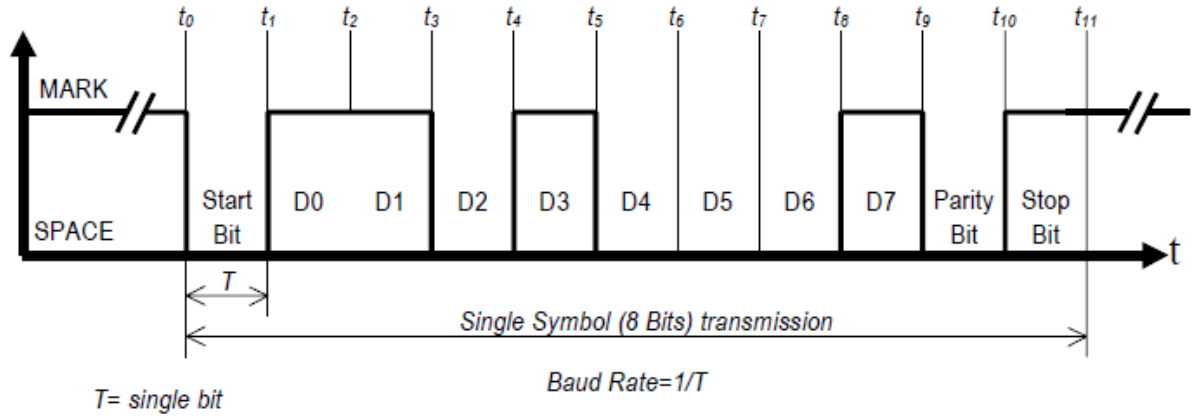



Figure 6

As seen in Figure 6, the idle state for the transmitter is logic high. When a transmission is going to be started, the Tx line is cleared for one time cycle. After this “start bit” has been sent, the data is loaded onto the bus in sequential order, starting with the least significant bit. After the data has been sent, a parity bit is sent to help with error checking. The parity bit is obtained by XORing all of the data bits together. After the parity bit is sent, the transmitter raises the bus back to logic high, and waits for the next transmit instruction.

The receiver works by constantly checking the Rx line. If it detects a falling edge on the wire, it waits half of a time period. Then, it waits a full time period before sampling the data. This is repeated for nine times. The receiver checks for parity by XORing all of the data and the parity bit together. If the output from this operation is zero, none of the bits got flipped, and the receiver outputs the data. If there was an error, a flag is toggled.

The UART is a good method of communication for low speed applications and simplicity.

## Module Details

| PORT NAME | DESCRIPTION |
|---|---|
| *sysclk_I* | Input port for main system clock |
| *rst_n_I* | Active low reset pin |
| *rx_enable_I* | Enables the receiver module to receive incoming data |
| *tx_enable_up_I* | Commands the transmit module to send the data present on *tx_data_bus_I* |
| *[7:0]tx_data_bus_I* | 8 bits of data that will be sent by transmitter |
| *rx_data_read_* | Control signal to tell the receiver that the data has been read |
| *[7:0]rx_data_bus_O* | Output data from the receiver module |
| *rx_data_ready_O* | Flag to signal when there is new data to be read from the receiver |
| *rx_error_O* | Flag to signal when there is an error in receiving a transmission |
| *tx_busy_O* | Flag to signal when the transmitter is in the middle of a transmission |
| *rx_input_I* | Rx input to the receiver module |
| *tx_output_O* | Tx output from the transmit module |

### Configurable Options

Setting the Baud Rate

The baud rate can be set to any desired value. It can be configured by changing the *baud_generator* module in the rtl_design/uart folder. Simply change the value of *divider1* based upon the comments provided in the code.

## Module Operations

The operations described here are simply a more readable version of the demo routines provided with the software.

### Sending a Byte

To send a byte, first load the data that is desired to be sent onto the tx_data_bus_I register of the UART. Next, check the tx_busy flag to make sure the UART isn't in the middle of a current transmission. Finally, toggle the tx_enable pin high then low. Wait until the UART initializes, and then you are free to go do something else.

### Receiving a Byte

To receive a byte, first check the rx_data_ready_O pin. If it is high there is new data present on the rx_data_bus_O. Read it into another variable, then toggle the rx_data_read line high then low.

It is recommended to either check the rx_data_ready_O pin using a processor interrupt that triggers at a slightly faster rate than the baud rate/11. This ensures that the receiver is checked every time a possible receive operation could be performed. A more efficient way of doing this would be to interrupt off of the rx_data_ready_O pin itself.

# SMBUS

## Theory of Operation

SMBus is shorthand for System Management Bus. SMBus is a serial communication method derived from $I^2C$ by Intel in 1995. It is used for low bandwidth communication between various integrated circuits, sensors and especially smart batteries.

SMBus differs from other transmission schemes in that it is a two way communication bus. It uses one wire for the clock and another wire for the data. Both of the wires are connected to VDD through a pull up resistor. Devices attached to the SMBus are normally in a high impedance state. When a device wishes to transmit, it either lets the bus get pulled high, or it uses an open drain transistor to pull the bus down to VSS. An example of a SMBus system is shown in Figure 7.

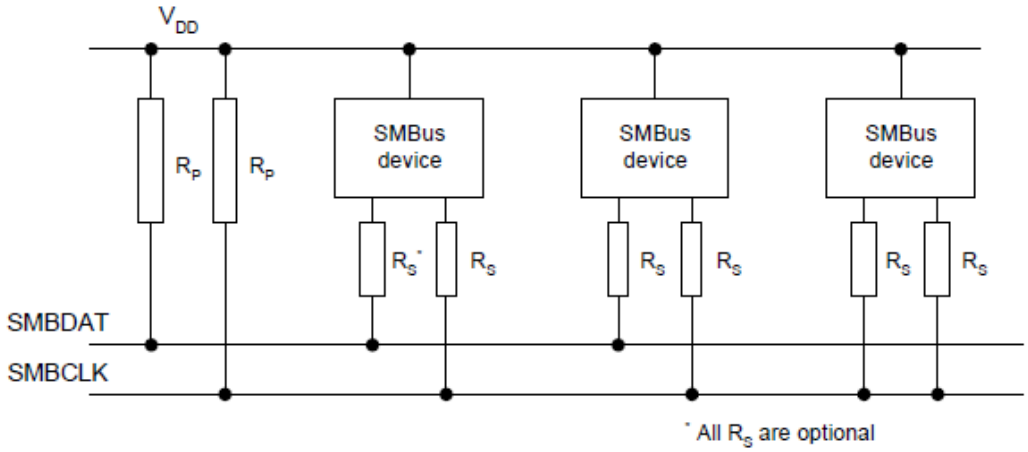



**Figure 3-4: SMBus branch with multiple devices attached**

Figure 7

There are two types of devices in a SMBus system, master and slave devices. Master devices initiate transactions, and drive the clock. Slave devices monitor the bus, and if they are addressed correctly they will respond using an acknowledge signal. The slave device will then respond according to the master's commands.

There are several different types of transactions that can take place between and SMBus master and slave. They are: read byte, write byte, write block and read block. In our SMBus master we only designed for single byte reads and writes, although the write block and read block functionality could be added in relatively easily without a lot of modification.

The SMBus protocol calls for several unique events that signal important changes in transmissions. These are start, stop, ACK and NACK events.

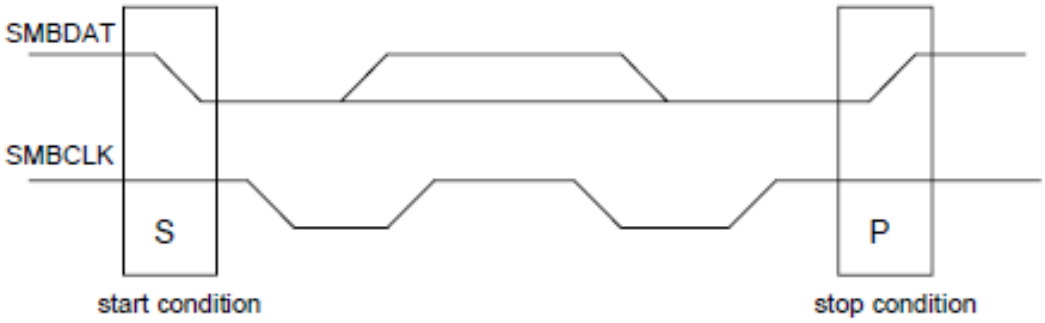



Figure 8

Figure 8 shows a start and stop condition. A start condition is used at the beginning of a transaction to signal devices to be ready to accept a transmission. Stop bits are used at the end of a transaction to signal devices that the transaction has been completed, or an error occurred and it had to be ended early.

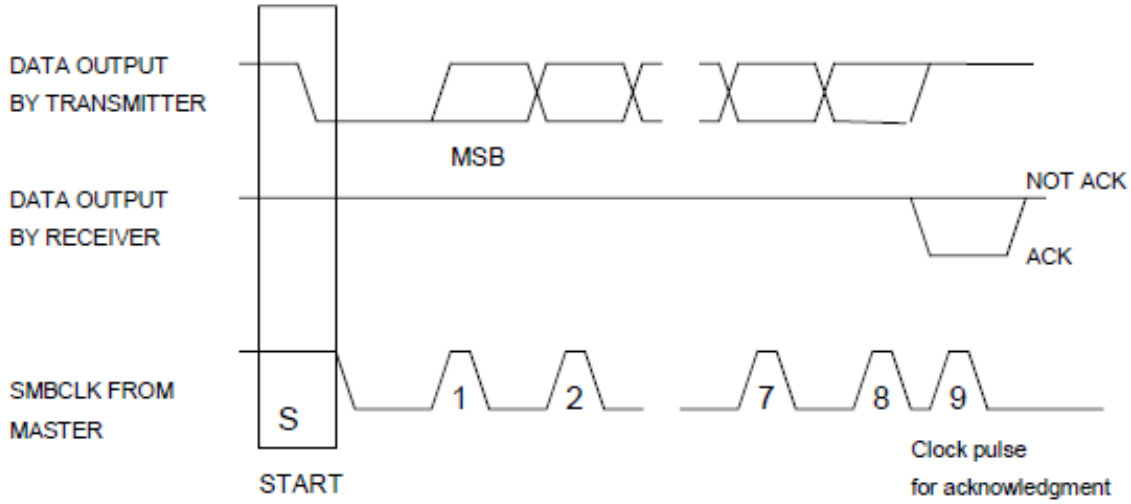



Figure 9

Figure 9 shows an acknowledge/not acknowledge signal. After 8 clock pulses, if the slave address has received the data correctly, it will pull the data line low for one clock cycle to signal the transmitter that it is working correctly. If a device does not receive the data correctly, it will let the data line float, and the transmitter will be able to recognize that something isn't working correctly.

As mentioned earlier, there are two protocols that our SMBus uses for dealing with data. The first protocol is writing a byte.

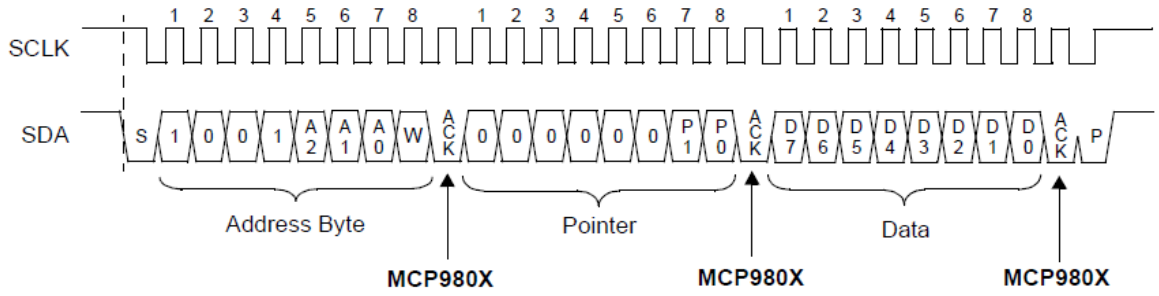



Figure 10

As we can see from Figure 10, the master starts by sending the address of the device. If there is a device present and it receives the transmission correctly, it will respond with an ACK. The master then sends the register address to write to. The device again ACKs if it received it correctly. Finally, the master sends the data that it wishes to write to the SMBus. The device ACKs again, and then the master sends a stop bit.

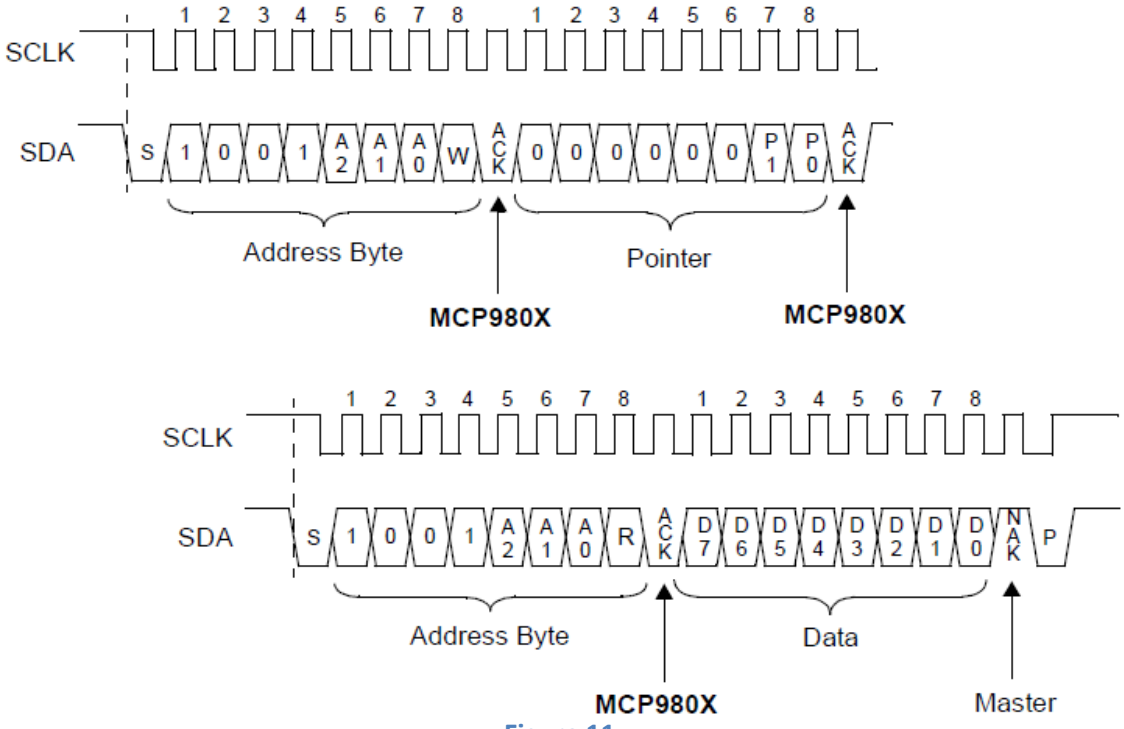



Figure 11

In Figure 11 we can see a read transaction. The master first sends the device address, and the device ACKs. Next, the master sends the address of the register to read from, and the device ACKs again. The master then immediately sends a start bit, followed by the address of the device. Finally, the device loads the requested data onto the bus, and the master reads it in sequential order. The master then sends a NACK when it has received the data.

## Module Details

| PORT NAME | DESCRIPTION |
|---|---|
| *SYSCLK_I* | Input port for main system clock |
| *RST_N_I* | Active low reset |
| *[7:0]DATA_I* | Input port for data to be sent on SMBus |
| *[7:0]SLAVE_ADDR_I* | Input port for slave device address |
| *[7:0]COMMAND_CODE_I* | Input port for slave device command/register address |
| *EN_I* | Enable transmission |
| *SCLK_I* | Input from top level SCLK |
| *SDAT_I* | Input from top level SDAT |
| *DATA_READ_I* | Input port to signal when data has been read |
| *[7:0]DATA_O* | Output port for data received from slave |
| *IDLE_O* | Flag that signals no transaction is in progress |
| *DATA_READY_O* | Flag that signals there is new data present |
| *SCLK_O* | Output to top level SCLK |
| *SDAT_O* | Output to top level SDAT |
| *ERROR_O* | Error in transmission |

### Configurable Options

Setting the Baud Rate

Open the sclk_gen.v file and change the timeout according to the instructions in the code

## Module Operations

### Sending a Byte

First, check the IDLE_O flag to make sure that the SMBus is not in the middle of another transaction. If it is high, load the slave address, command code and data onto their respective registers. Next, toggle EN_I high then low. Wait until the idle line comes high, and then check the ERROR_O pin. If it is low, the transmission completed successfully.

### Receiving a Byte

First, check the IDLE_O flag to make sure that the SMBus is not in the middle of another transaction. If it is high, load the slave address and command code. Then, toggle EN_I high then low. Wait until the IDLE_O line returns to high. When it does, check the ERROR_O and DATA_READY_O flags. If there isn't an error, get the data. If ERROR_O is high, there will be data on the output bus, but it may be corrupt or useless.

## LCD Initialize

### Theory of Operation

We had several reasons for wanting to implement a hardware controller for an LCD screen. First of all, writing to an LCD is very time consuming for a processor, since there are long waits and timing protocols that must be followed. This means that without some very complex code, the processor will be sitting idle waiting for the LCD to take its precious time. This is time that could be spent doing other more useful things, or going into a low power state. We thought that by taking care of these steps in hardware that we could simplify the programmer's life, and it would also give us experience designing something that is a little bit more obscure.

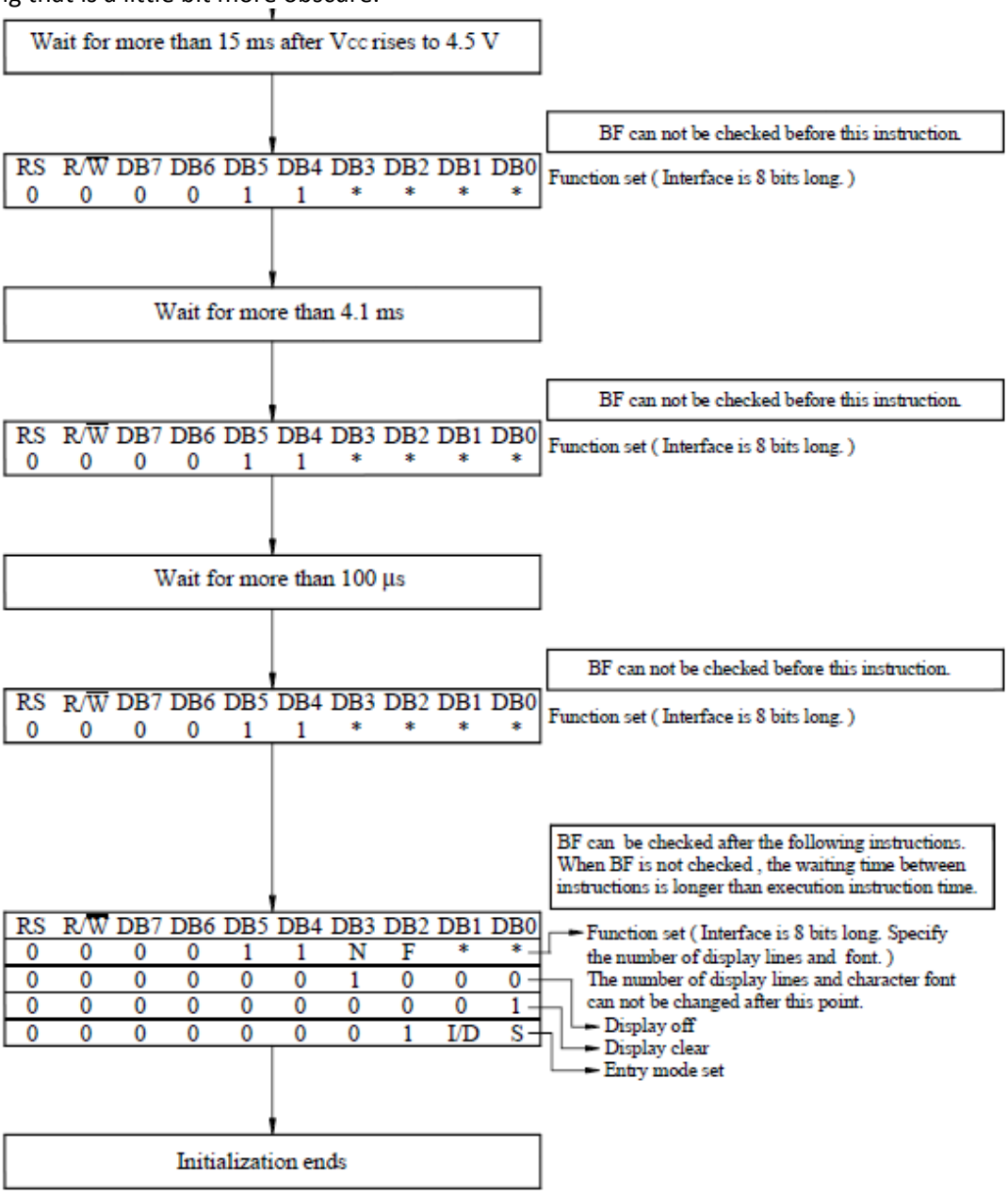



Figure 12

The LCD has several sets of protocols for how commands must be sent to it. There are two sets of commands. One set is executed after the LCD is powered up, and it configures the desired options for the LCD operation. We built one block to take care of this initial powerup, and its set of operations is defined in Figure 12. The other set of operations deals with writing normal characters to the screen, moving the cursor, etc. The lcd_controller handles all of these operations. These operations are defined in Figure 13.

| Instruction | Instruction Code | | | | | | | | | | Description | Execution time (fosc=270Khz) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | RS | R/W | DB7 | DB6 | DB5 | DB4 | DB3 | DB2 | DB1 | DB0 | | |
| Clear Display | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | Write "00H" to DDRAM and set DDRAM address to "00H" from AC | 1.53ms |
| Return Home | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | — | Set DDRAM address to "00H" from AC and return cursor to its original position if shifted. The contents of DDRAM are not changed. | 1.53ms |
| Entry Mode Set | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | I/D | SH | Assign cursor moving direction and enable the shift of entire display. | 39 μs |
| Display ON/OFF Control | 0 | 0 | 0 | 0 | 0 | 0 | 1 | D | C | B | Set display (D), cursor (C ), and blinking of cursor (B) on/off control bit. | 39 μs |
| Cursor or Display Shift | 0 | 0 | 0 | 0 | 0 | 1 | S/C | R/L | — | — | Set cursor moving and display shift control bit, and the direction, without changing of DDRAM data. | 39 μs |
| Function Set | 0 | 0 | 0 | 0 | 1 | DL | N | F | — | — | Set interface data length (DL:8-bit/4-bit), numbers of display line (N:2-line/1-line)and, display font type (F:5 × 11 dots/5 × 8 dots) | 39 μs |
| Set CGRAM Address | 0 | 0 | 0 | 1 | AC5 | AC4 | AC3 | AC2 | AC1 | AC0 | Set CGRAM address in address counter. | 39 μs |
| Set DDRAM Address | 0 | 0 | 1 | AC6 | AC5 | AC4 | AC3 | AC2 | AC1 | AC0 | Set DDRAM address in address counter. | 39 μs |
| Read Busy Flag and Address | 0 | 1 | BF | AC6 | AC5 | AC4 | AC3 | AC2 | AC1 | AC0 | Whether during internal operation or not can be known by reading BF. The contents of address counter can also be read. | 0 μs |
| Write Data to RAM | 1 | 0 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | Write data into internal RAM (DDRAM/CGRAM). | 43 μs |
| Read Data from RAM | 1 | 1 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | Read data from internal RAM (DDRAM/CGRAM). | 43 μs |

Figure 13

## Module Details

**Port List**

| ***Port Name*** | **Port Function** |
|---|---|
| *[7:0]lcd_data* | Output data to the LCD screen |
| *lcd_rw* | Output read/write select |
| *lcd_rs* | Command/data select |
| *lcd_en* | Enable the LCD |
| *lcd_on* | Turn LCD on/off |
| *lcd_blon* | Turn LCD backlight on or off |
| *mclk* | Main system clock |
| *rst* | reset |
| *[7:0]lcd_char* | Character to be written to the LCD |
| *[7:0]lcd_loc* | Location to write the character to |
| *[7:0]lcd_config* | Registers to configure different options |

**Lcd_config Register**

| | | |
|---|---|---|
| rw | lcd_config[0] | Pass through signal |
| rs | lcd_config[1] | Pass through signal |
| en | lcd_config[2] | Pass through signal |
| on | lcd_config[3] | Pass through signal |
| blon | lcd_config[4] | Pass through signal |
| save | lcd_config[5] | Save new data to be written to the LCD |
| bypass | lcd_config[6] | This register allows commands to bypass the module and write to the LCD |

## Module Operations

The LCD Interface is very easy to control. Simply place the desired character and location to be written to on the lcd_char and lcd_loc registers and toggle lcd_config[5] high then low.

If desired, lcd_config[6] can be set, and this allows the lower bits in lcd_config to pass directly through to the LCD, allowing it to be controlled manually if desired.

The LCD initialize and controller blocks have somewhat complicated state machines. The state diagrams can be viewed on the next pages.

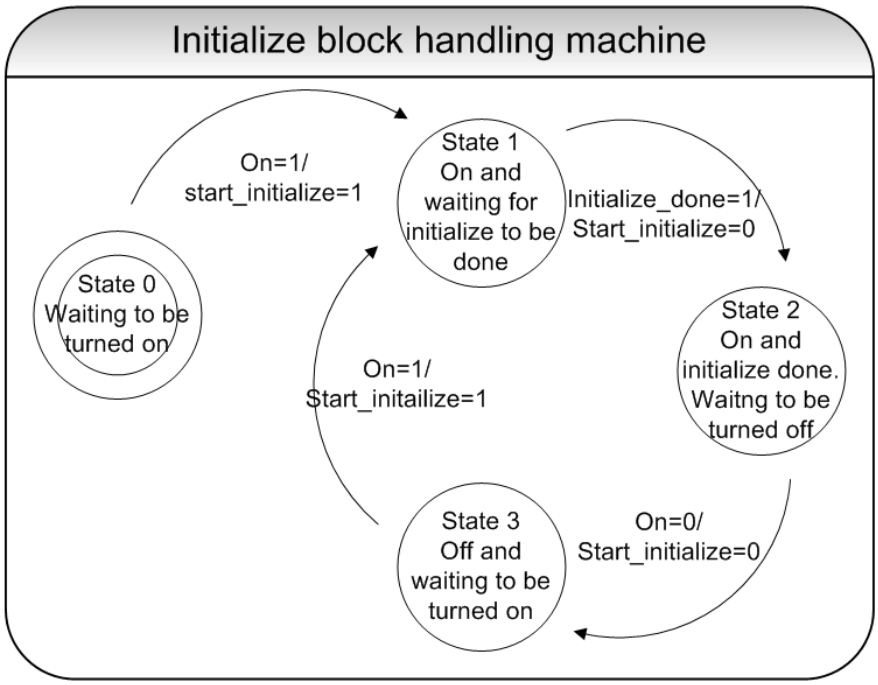



Figure 14

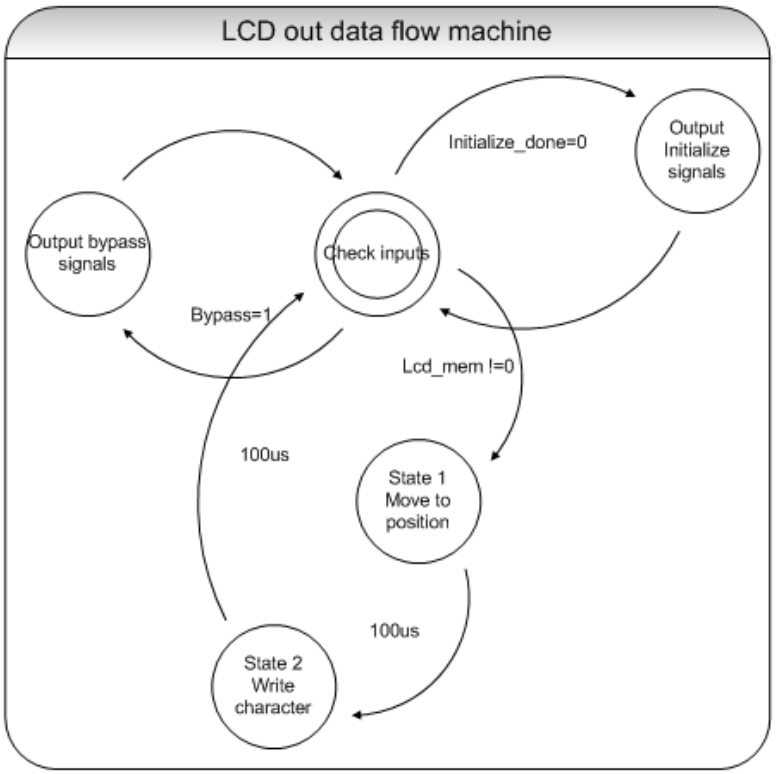



Figure 15

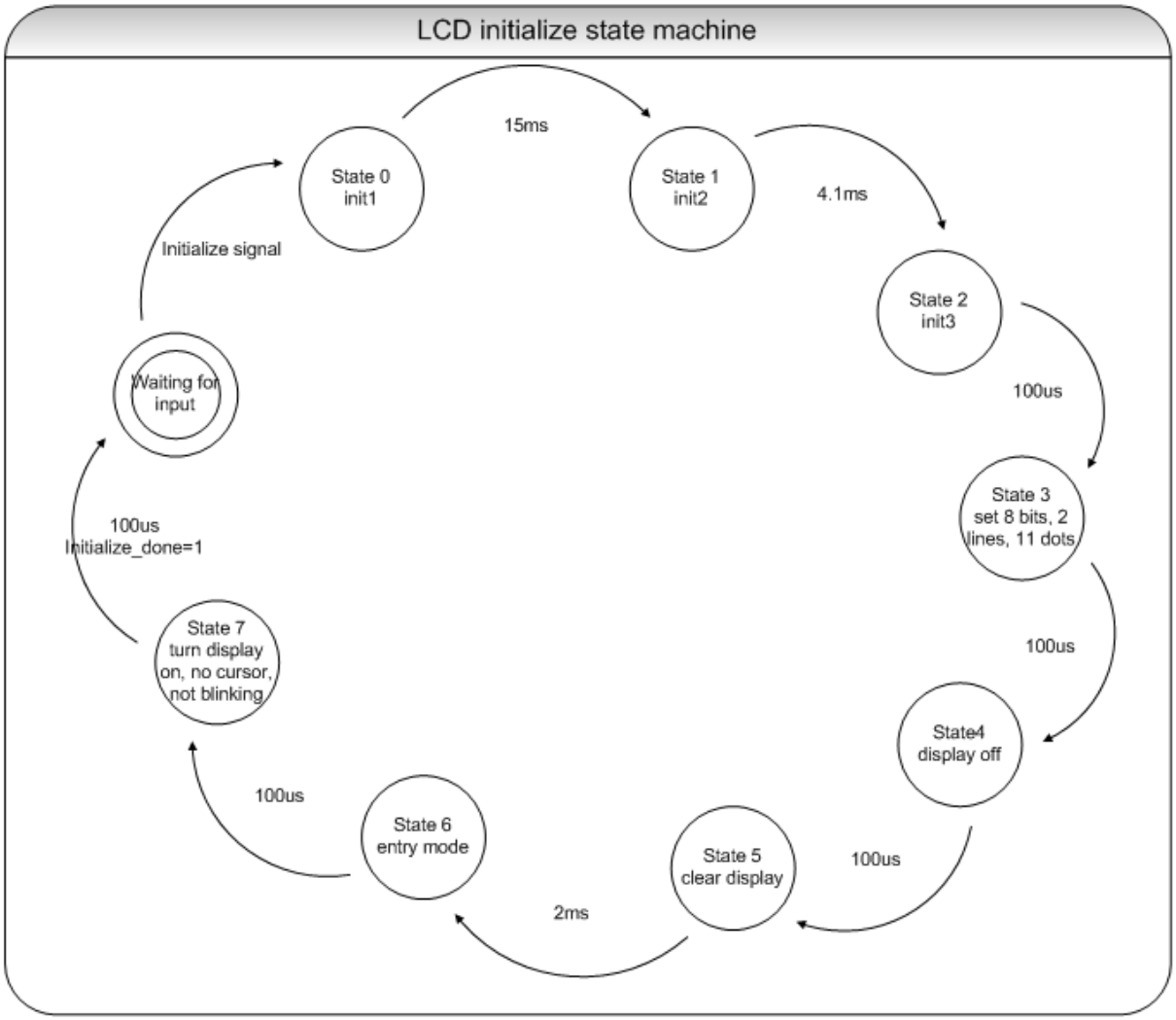



Figure 16

## File Structure

Our file structure for the project is rather large and complex, and thus is included on the project CD since it does not lend well to being printed.

# Software

There were two different sets of software involved in our project. The first set is what ran on the microprocessor and carried out our operations there. The second set ran on a host computer and allowed us to communicate with our microprocessor using a serial port.

## MSP430

Our MSP430 software served to handle all of our data gathering and storing routines, as well as communicating with the UART, LCD Interface and SMBUS. We have included function names as well as a brief description of what task they accomplish. Also included are flowcharts for main algorithm operations, and a brief memory map of important registers.

### Function Descriptions

| Function Name | Function Description |
|---|---|
| void lcd_write_char(int loc, char *character, int length) | Takes an integer from 0 to 31, a reference to a character string, and the number of characters to output |
| void lcd_clear (void) | Clears the LCD screen |
| void smbus_write (char slave_addr, char command, char data) | Takes a slave address, command code, and data byte and writes it to the specified SMBus device |
| char smbus_read (char slave_addr, char command) | Returns a byte from the specified device and register |
| void leds_off(void) | Turns all of the demo board LEDs off |
| void led0_on(void) | Turns on LED0 on the demo board |
| void led1_on(void) | Turns on LED1 on the demo board |
| void led2_on(void) | Turns on LED2 on the demo board |
| void led3_on(void) | Turns on LED3 on the demo board |
| int temp_read(void) | Returns the temperature from the temp sensor |
| mem_read(int address) | Reads from the desired memory address |
| mem_write(int address, char data) | Writes data to the desired memory address |
| void led_showtemp(int temp) | Updates the LEDs to match the current temp |
| void uart_send (int n, char * word) | Sends a byte of data on the UART |

### SMBus Device Addresses

| SMBus Device | Address |
|---|---|
| LED_ADDR_W | 0xC4 |
| LED_ADDR_R | 0xC5 |
| LED_PSC0 | 0x01 |
| LED_SEL | 0x05 |
| TEMP_ADDR_W | 0x9E |
| TEMP_ADDR_R | 0x9F |
| EEPROM_ADDR | 0x0A |

## MSP430 Memory Map

| Register | Address |
|---|---|
| TXDATA_ | 0x0094 |
| RXDATA_ | 0x0095 |
| UARTOUT_ | 0x0096 |
| UARTIN_ | 0x0097 |
| SM_SLAVE_ | 0x0098 |
| SM_REG_ | 0x0099 |
| SM_DATA_I_ | 0x009A |
| SM_DATA_O_ | 0x009B |
| SMBUS_IN_ | 0x009C |
| SMBUS_OUT_ | 0x09D |
| LCD_CHAR_ | 0x00A2 |
| LCD_LOC_ | 0x00A3 |
| LCD_CONFIG | 0x00A4 |

## Peripheral Flags/Configuration Bits

| Peripheral Flag/Config | Hex |
|---|---|
| rx_data_ready | 0x04 |
| rx_error | 0x02 |
| tx_busy | 0x01 |
| rx_data_read | 0x04 |
| tx_enable | 0x02 |
| rx_enable | 0x01 |
| smbus_en | 0x01 |
| smbus_data_read | 0x02 |
| smbus_reset | 0x04 |
| smbus_error | 0x04 |
| smbus_idle | 0x02 |
| smbus_data_ready | 0x01 |
| rw | 0x01 |
| rs | 0x02 |
| en | 0x04 |
| on | 0x08 |
| blon | 0x10 |
| save | 0x20 |
| bypass | 0x40 |

## Flowcharts for Software Operations

**MSP430 Main Routine**

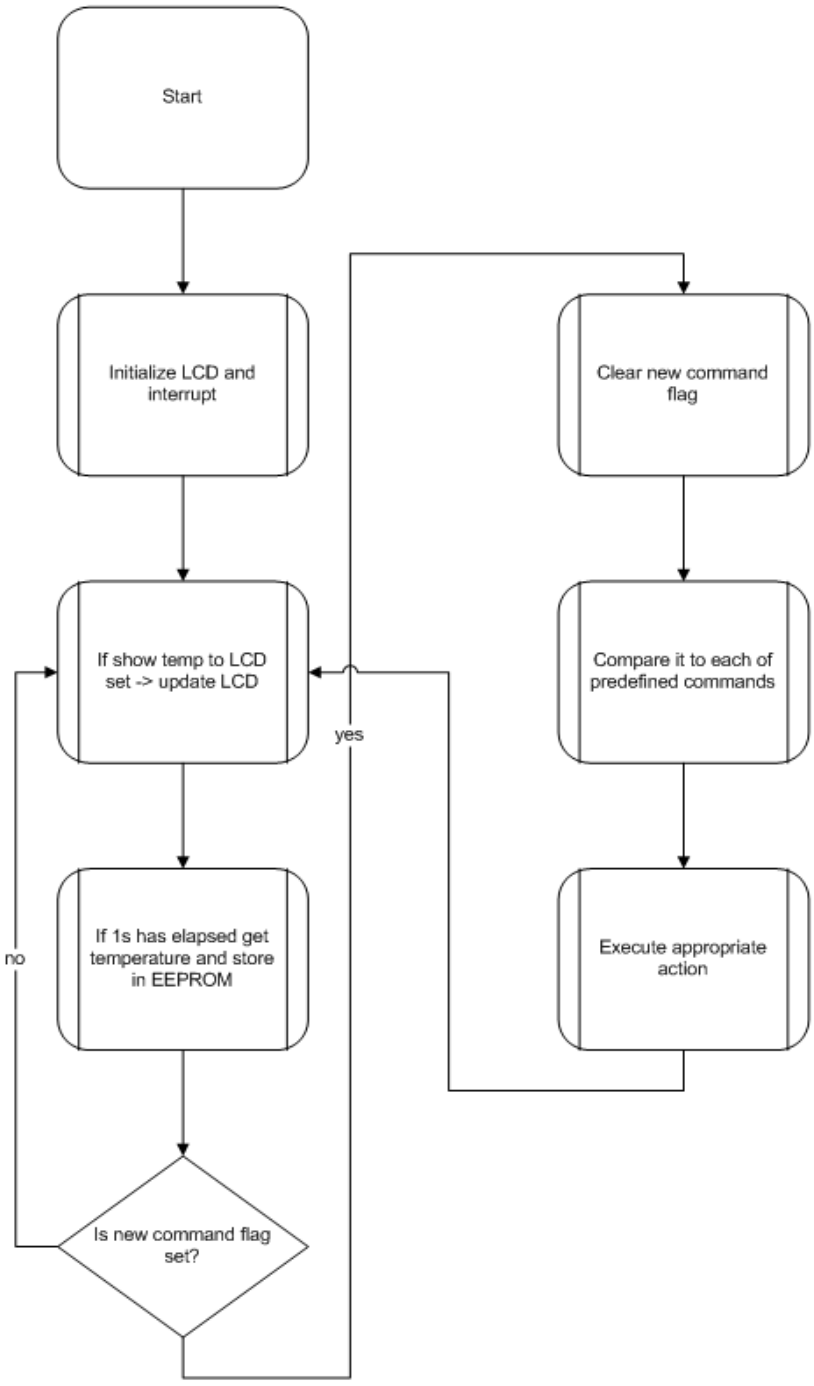



Figure 17

**MSP430 Interrupt Service Routine**

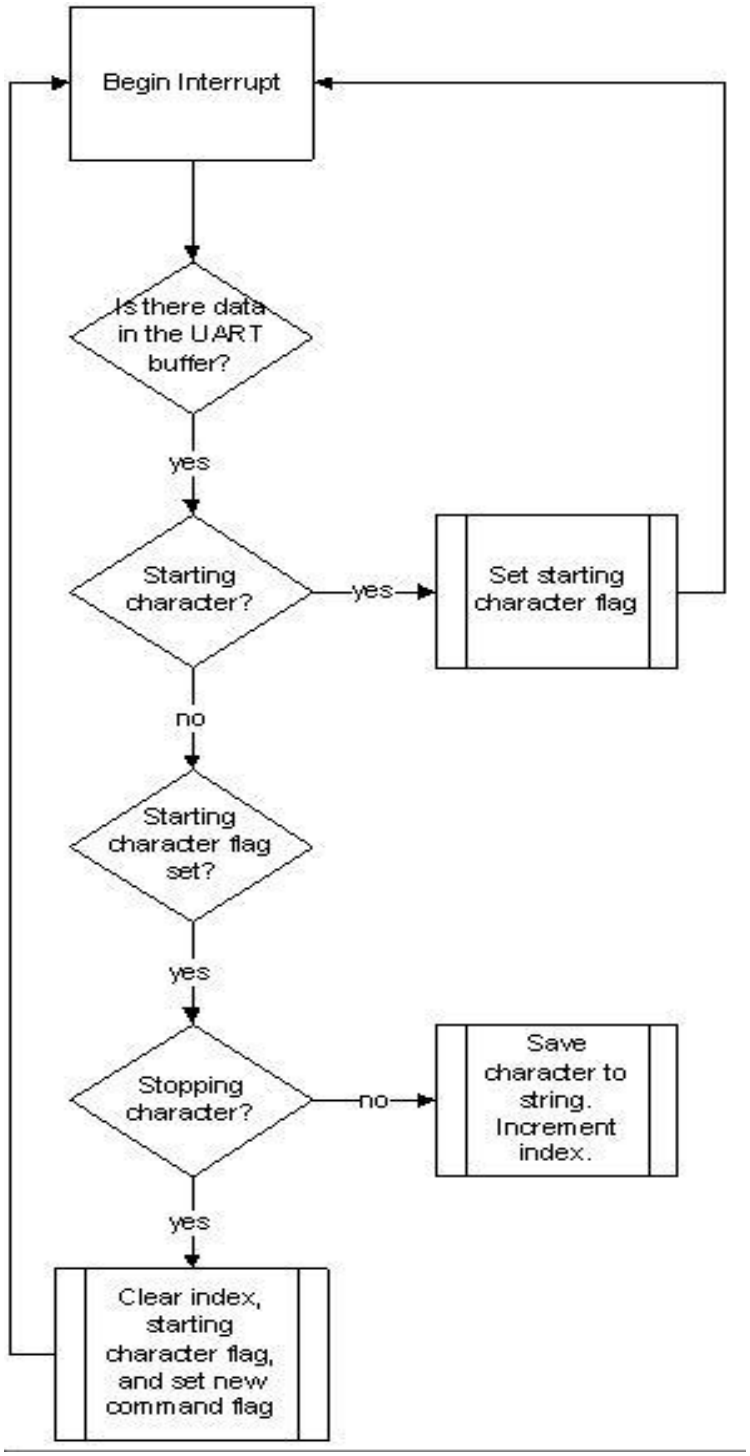


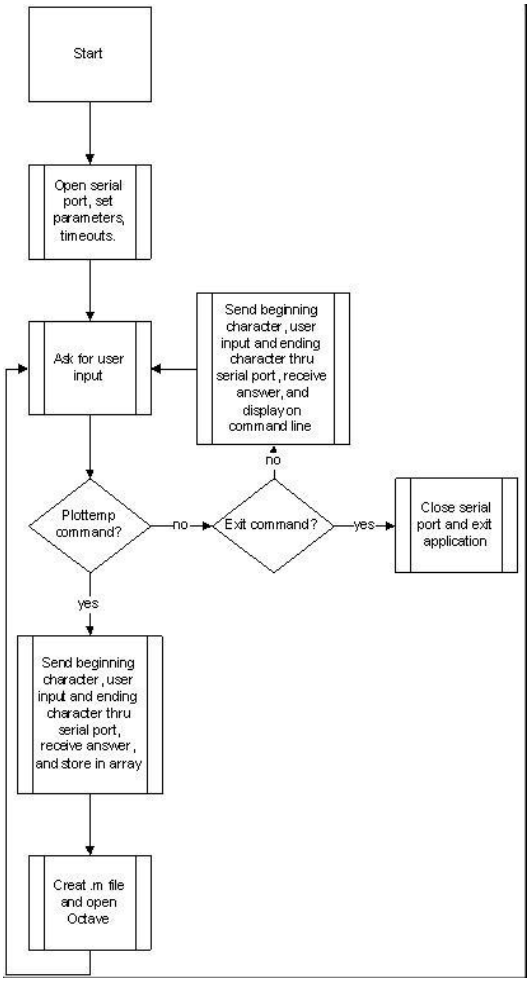

Figure 18

## PC Software

The PC software was created to allow us to access data and functionality on the demo system by using a command line interface. All of our code was written in C++, and we have included flowcharts, function descriptions and other useful information here.

| Function Name | Function Description |
|---|---|
| lcd_clear | Clears the lcd screen |
| lcd_write [message] | Writes [message] to the DE2 lcd screen beginning from first line, first spot |
| lcd_off | Turns the lcd screen off |
| lcd_temp | Displays temperature information on lcd screen |
| temp | Displays current temperature on PC |
| plottemp | Displays a plot of the temperature over the last minute on PC using Octave (data read from EEPROM) |
| string[n] | Displays custom message [n] on PC |

**PC Software Routine**



# Demo PCB

**Purpose:**

The purpose of the PCB is to demonstrate that the SMBus we designed does work and can be used in a real life application. The real life application demonstrated is a data gathering platform that can record the temperature.

The PCB has three SMBus devices on it, a temperature sensor, 16kb of EEPROM and an LED driver. They are linked to the SMBus though zero ohm resistors, which allow the connection or disconnection of any device in order to allow for debugging.

**Circuit Diagram:**

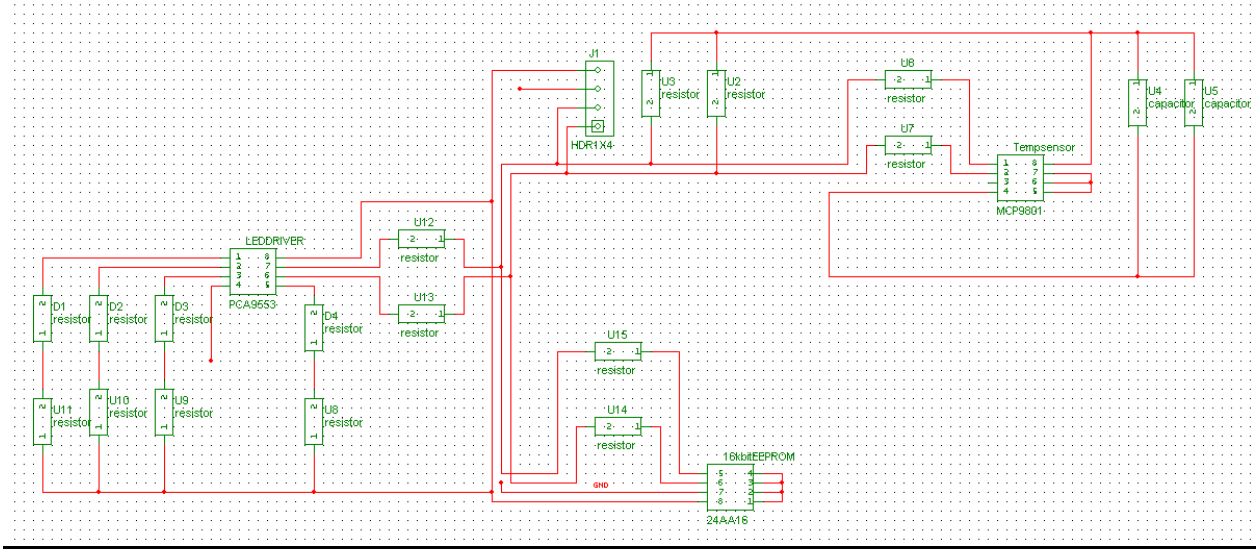
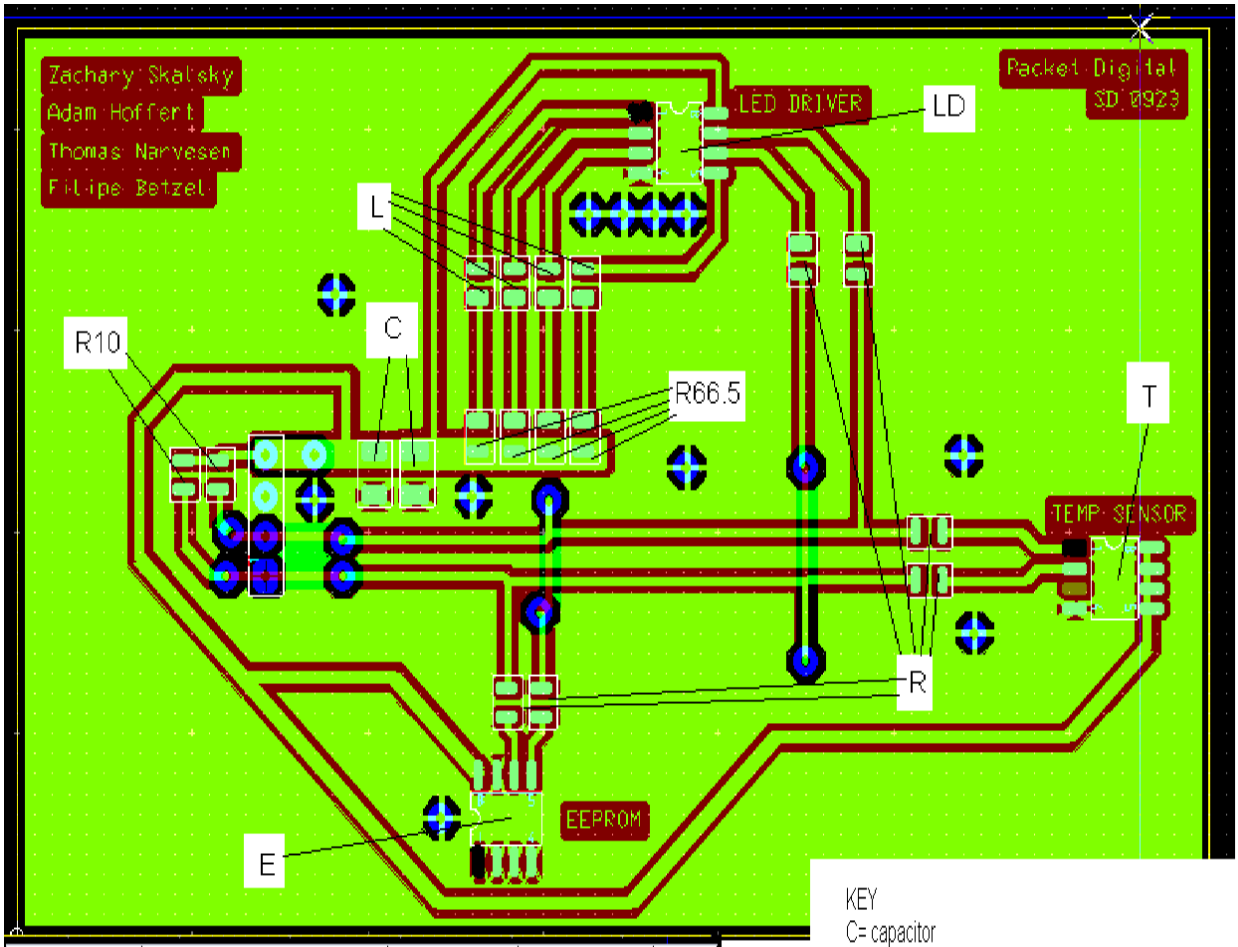



Figure 19




| Part | Part Description | Digikey Part # | Manufactor Part # | Quantity |
|---|---|---|---|---|
| Capacitor | CAP 10UF 6.3V CERAMIC X5R 1206 | PCC1940CT-ND | ECJ-3YB0J106K | 2 |
| Resistor | RES 10.0K OHM 1/8W 1% 0805 SMD | P10.0KCCT-ND | ERJ-6ENF1002V | 2 |
| Resistor | P66.5CCT-ND | P66.5CCT-ND | ERJ-6ENF66R5V | 4 |
| Resistor | RES 0.0 OHM 1/8W 0805 SMD | RMCF1/100RCT-ND | RMCF 1/10 0 R | 6 |
| LED | LED GREEN CLEAR 0805 SMD | 160-1414-1-ND | LTST-C170KGKT | 4 |
| LED Driver | IC LED DRIVER LINEAR 8-SOIC | 568-3390-5-ND | PCA9553D/01,112 | 1 |
| 16kbitEEPROM | IC EEPROM 16KBIT 400KHZ 8SOIC | 24AA16-I/SN-ND | 24AA16-I/SN | 1 |
| Temperature Sensor | IC SENSOR THERMAL 2.7V 8SOIC | MCP9801-M/SN-ND | MCP9801-M/SN | 1 |

KEY
C= capacitor
R10= 10k ohm
R66.5= 66.5 ohm
R= 0 ohm
L= LED
LD= LED Driver
E= EEPROM
T= Temperature sensor

Pin number 1 on the LED Driver, EEPROM, and Temp sensor will be marked black

**Figure 20**

**Size and dimensions:**

The board is 3401 mills by 2252 mills. The smallest traces are 30 mills.

Gerber files are included on the project CD for building the PCB. Also included are the PCB schematic and layout.

**SMBUS communication to other devices**

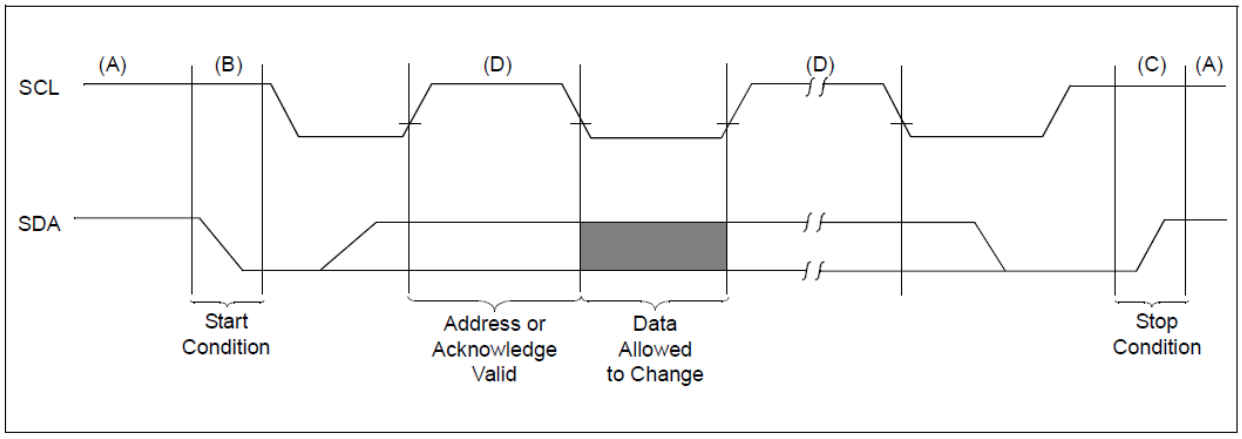



Figure 21

The diagram above shows how the SMBUS sends or receives data. The first rule you will notice is that the SCL must be high in order to send or receive data on the SDAT. A falling edge is a start condition and a rising edge a stop condition. If you want to change data you need to do so when the SCL is low to avoid getting a start or stop condition.

**EEPROM**

Write to EEPROM

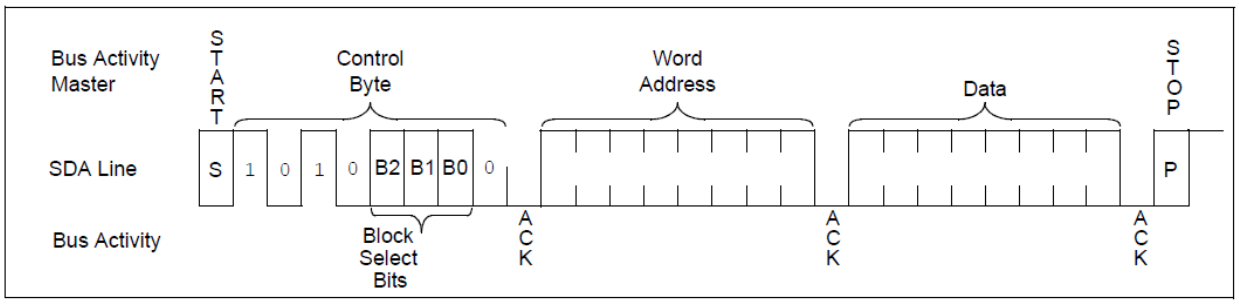



Figure 22

The diagram above shows what is happening on the SDAT line when the SMBUS is writing to it. The first bit is the start bit

**LED Driver**

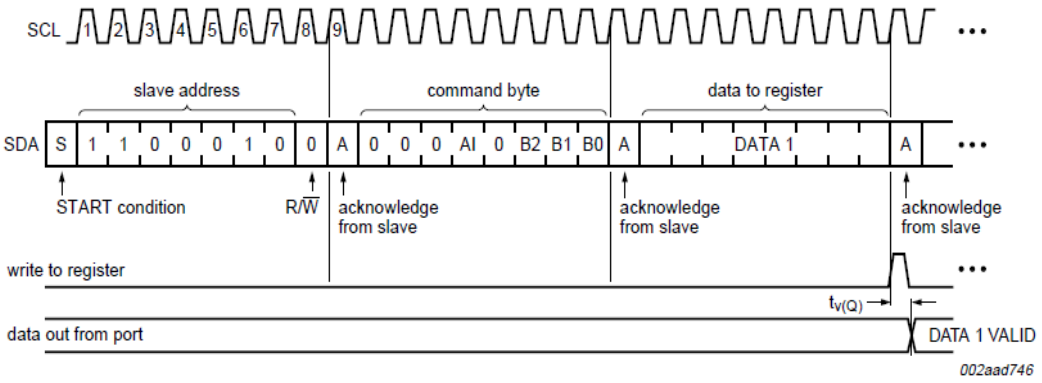



Figure 23

The first byte is the slave device address and also tells you if you are doing a read or a write (0 for write 1 for read). The second byte is the command byte and is used to decide if you want just LED's on or pulse width modulation. There are 6 different functions and a function is set by B0, B1, and B2. The last byte is the data byte and is used to set the PWM or select which LED's are on.

**Temperature Sensor**

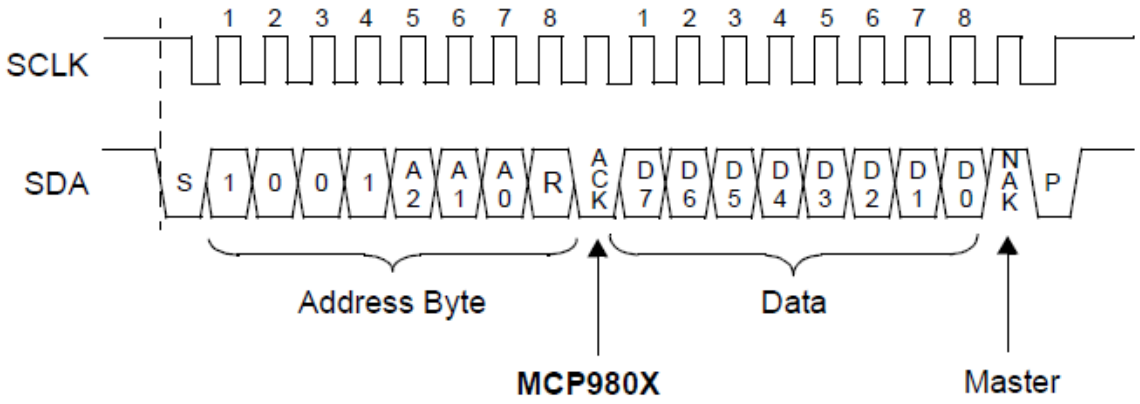


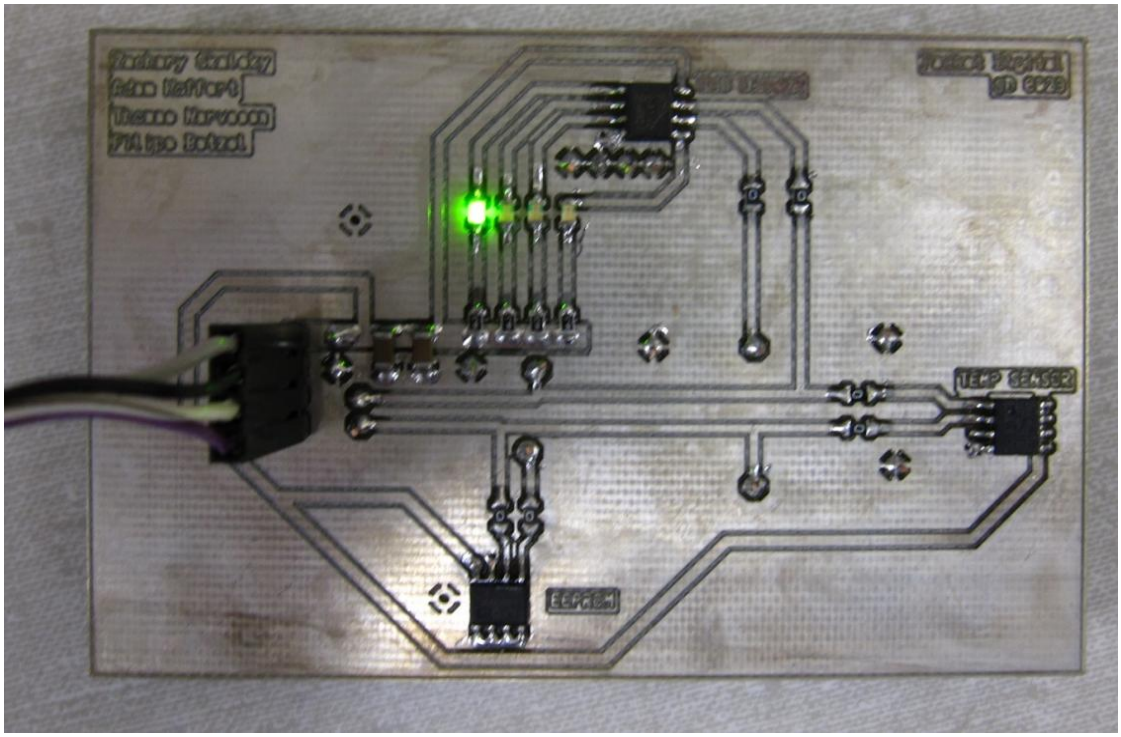

Figure 24

We read from the temperature sensor the way the diagram above shows. The first byte is the address and the second byte is reading the data.

**Revision 2 Changes**

- Made header able to be solder as a threw hole device instead of surface mount
- To prevent heat loss to the soldering iron during assembly, we put copper X's on our pads instead of solid copper. This was mostly done to the pads that were connected to the ground plane.
- Moved some traces and made them wider.
- Added some via's for better ground plane connectivity.

## Picture of Assembled PCB



# Troubleshooting

**Troubleshooting the PCB**

1. Check that the cables are attached to the correct pins on the FPGA and the demo PCB.
2. Do a quick check of solder connections to make sure that parts have not been damaged
3. Check the power to ground and bus to ground voltages
    a. Power to ground should be 3.3V
    b. If the SMBus is idle, the bus voltages should be about 3.3V
    c. If the voltages are not what they should be, disconnect power and check for power to ground or bus to ground shorts using an ohmmeter.
4. If an oscilloscope is available, put the MSP430 into a data gathering mode and probe the SMBus. Data transactions should be taking place, and they can be viewed to make sure that the respective devices are responding
5. Remove the zero ohm resistors on the SCLK and SDAT lines to all but one device, and run a test to make sure that the device is working correctly. Once operation has been verified, begin connecting the other devices.

**Troubleshooting the MSP430**

Due to the complex nature of the software, there isn't a lot that can be done without taking time to understand the source code. However, there are a few things that can be checked if the MSP430 is not operating properly

1. Power cycle the DE2 and reprogram the FPGA using the Quartus programmer and main.sof
2. Recompile the MSP430 software and load it onto the FPGA using Quartus
3. If the serial port interface isn't working, check to make sure that the correct serial port is defined in the PC application code.
4. Check to make sure that the PCB is correctly attached. If the MSP430 is trying to talk to devices that aren't there, it's possible that it may hang or cause unexpected operation

# Project Comments

### Prior Knowledge

A working knowledge of hardware definition languages would have been a huge benefit in working on a project like this. We spent a couple of months just getting proficient in writing and simulating Verilog. Digital System Design and Implementation would be a good class to take, even though it covers VHDL instead of Verilog. Many of the concepts are the same however, and transitioning to Verilog would require only learning some new syntax. Another helpful class to have taken would have been Computer Organization. It provides at least an introduction to how microprocessors are built, and would have made the learning curve a lot less steep.

### Project Issues

We believe that our product is very stable, although there are a couple more things that could be done in order to improve this. First, incorporate a watchdog timer into the MSP430 just in case something goes wrong at it hangs. A watchdog timer would reset the processor and hopefully restore operation. Second, get the status flags working on the SMBus. Right now the project is using wait statements when performing SMBus transactions, but if the status flags were working it would be much faster to check them. The flags are working on the lower levels of the SMBus, but for some reason they are not showing up in the MSP430 registers. It could be an issue with the register definition in the smbus_msp430.v file, but we checked and double checked everything in there. Perhaps a new address would solve the problem.

### Future Improvements

Some other improvements that could be made would be to increase the buffer size in the UART to 8 or 16 bytes. Right now we are at 1 byte, which means that the processor has to check the buffer very often in order to make sure that data is not written over. A larger buffer would reduce processor overhead, since it could check it less frequently.

Also, more user configurable options and features could be added to the UART and SMBus. For example, being able to set the baud rate for the UART and SMBus, stop and parity bit enables for the UART and multibyte reads and writes as well as packet error checking for the SMBus. It would also be helpful to be able to easily configure the size of the LCD screen.

### Derived Projects

We see our device as being very useful for data gathering purposes. If built into a real chip, our custom microcontroller would provide low power and high flexibility with the ability to integrate into many different devices through the use of the SMBus and UART. An example of this would be integration with the weather station. We would be able to gather and store data using only our controller, and would be able to eliminate having a PC hooked up to the station at all times.

# Appendix

## Parts List

These parts were used to make The PCB

| Part | Digikey Part # | Manufactor Part # | Quantity | Price per unit | Price total |
|---|---|---|---|---|---|
| Capacitor | PCC1940CT-ND | ECJ-3YB0J106K | 8 | 0.49 | 3.92 |
| Resistor | P10.0KCCT-ND | ERJ-6ENF1002V | 20 | 0.091 | 1.82 |
| Resistor | P66.5CCT-ND | ERJ-6ENF66R5V | 20 | 0.091 | 1.82 |
| Resistor | RMCF1/100RCT-ND | RMCF 1/10 0 R | 24 | 0.021 | 0.5 |
| LED | 160-1414-1-ND | LTST-C170KGKT | 8 | 0.12 | 0.96 |
| LED Driver | 568-3390-5-ND | PCA9553D/01,112 | 3 | 1.79 | 5.37 |
| 16kbitEEPROM | 24AA16-I/SN-ND | 24AA16-I/SN | 3 | 0.4 | 1.2 |
| Temperature Sensor | MCP9801-M/SN-ND | MCP9801-M/SN | 3 | 1.34 | 4.02 |
| Total | | | | | $19.61 |

DE1 was supplied by Packet Digital and the DE2 was supplied by NDSU ECE department